Cache Simulator for Multiple Applications Running on a Single Core

Urvi Gada, Rajdeep Manjre, Prachi Kulkarni
Department Electrical and Computer Engineering
University of North Carolina at Charlotte

***Abstract*— This report summarizes the work undertaken, approach and results observed by simulating a unified cache for multiple applications running on a single core. The project analyzes the results observed by comparing between two and three applications running on a single core using First Come First Served (FCFS) and Round Robin (RR) scheduling algorithms. The cache mapping functionality used are Direct cache mapping and Fully Associative cache mapping as the cache was supposed to be analyzed for extreme cache mappings for effective results. Also, the cache replacement policy used for the analysis is First In First Out (FIFO) as it gives the least effective performance compared to LRU (Least Recently Used) replacement policy thus the cache simulator is analyzed in the worst condition possible. The analysis also compares the performance by varying the block size.**

Keywords— Cache replacement policy, Scheduling algorithms, First In First Out (FIFO), First Come First Served (FCFS), Round Robin (RR).

I. INTRODUCTION AND MOTIVATION

The main aim of our project is to build a cache simulator with a functionality to access multiple trace files i.e. multiple applications and analyze the performance of cache simulator when multiple applications are running on a single core and utilizing a single unified cache. Nowadays, response time is one of the most crucial factor in grading the performance of the computer architecture. Taking this fact into consideration we have implemented a cache simulator which uses the Round Robin scheduling algorithm (alike Preemptive scheduler) to reduce the response time of the scheduled applications. The Round Robin (RR) scheduler is compared with First Come First Served (FCFS) (alike Non-Preemptive) scheduler. In FCFS the applications are fetched sequentially one after the another while in RR each trace file is accessed simultaneously. The simulator is used to check how the Round Robin policy affects the cache utilization as compared to FCFS in extreme cache mapping polices i.e. Direct Mapping and Fully Associative Mapping. To check how robust is the built cache simulator we have used a FIFO (First In First Out) Replacement Policy which gives the least performance as compared to LRU (Least Recently Used) and Optimal Replacement Policy. Comparing Replacement policies is not included in this project. Also, it is approximated that all the trace files have unique addresses and no trace file has a similar memory address as the possibility of this happening is quite negligible. Furthermore, the block size is varied to check how the conflict misses affect the performance of the cache.

II. APPROACH

The cache simulator used for this project is a unified cache with size 32KB. The unified cache holds both instructions and data using a same instruction/data path via small register file. This was used to track the addresses of the trace files of applications that were used for the performance evaluation.

A. *Cache mapping strategies:*

The first parameter into consideration was for the cache mapping. The two cache mapping strategies used were the Direct mapped cache and Fully associative cache mapping. These two cache mapping strategies were chosen to check for the best approach of cache allocation that result into a better performance for multiple applications running on a single core. The idea behind direct mapped and fully associative cache is as shown below:

i) Direct mapped Cache:

Direct mapped caching, as the name suggests is a direct mapping technique where in each memory block is assigned to a specific line in the cache. The cache address is made up of three parts as shown in the diagram below –

TAG	INDEX	OFFSET
Identity for memory block	Number of cache blocks or line	Bytes within a block

The MSB bits represent the tag to be stored in line of cache to the corresponding to the block stored in line. The index bits represent the line where the block is stored. The LSB bits identify the word in each block. Direct mapping is easier to implement but the cache crashes if the blocks map to same line repeatedly, reloading the line over again thereby increasing the miss rates.

ii) Fully associative Cache:

Fully associative cache mapping technique assigns the memory block to any line in the cache. Since this technique provides flexibility of placing the memory block to any line in the cache, this technique provides an advantage of full cache utilization. The hit rate of fully associative cache is more as compared to other techniques, since increasing the associativity reduces the conflict misses. In case of cache misses, fully associative cache can provide flexibility of utilizing any cache replacement policy. However, this cache allocation policy is slower since the time to iterate through the lines is more. It also consumes more power and is costlier as the hardware implementation is complex.

It is observed that the cache hits of fully associative cache mapping strategy are better as compared to the direct mapped cache allocation strategy. This is because as the associativity of the cache is increased, the conflict misses decrease. This happens because in fully associative cache, the cache allocation is more flexible and cache utilization is optimum as compared to the direct mapped cache.

B. *Cache Replacement Policies:*

The next parameter into consideration was the cache replacement policy. The cache replacement policy used for the analysis of this project was the First In First Out (FIFO) replacement policy. This policy was used for this project to check for the “Worst Case” analysis when running multiple applications on a single core system. The basic idea behind a FIFO replacement policy is that the when the cache is full, the block that will be replaced is the block that had contained data the first amongst all the blocks. As compared to other replacement policies such as LRU and optimal replacement policy, FIFO is quite simple and easy to execute. However, this technique may not be able to exploit the spatial locality to the fullest and hence for this project it is considered as the worst-case analysis policy.

C. *Scheduling Algorithms:*

The third parameter into consideration is scheduling of the applications in the cache. The two scheduling algorithms used in this project are the Round Robin (RR) and First Come First Served (FCFS) scheduling algorithms. These two algorithms were chosen to check which of the two provide a better result. The basic idea of the two scheduling algorithms is as shown below:

i) First Come First Served (FCFS) Scheduling Algorithm:

FCFS abbreviated for First Come First Served is also called as First In First Out (FIFO) scheduling algorithm. It is a simple scheduling algorithm that processes an application as it arrives and runs it to completion before serving another application. The first process to arrive first is served first. The major drawback of this algorithm is that it is non-pre-emptive, making the processes with smaller execution time starve while the processes with larger bursts are served if they are scheduled earlier.

In this project, the FCFS algorithm exploits spatial locality to the fullest and hence the hit rate for FCFS will be more as compared to RR. This is because each application will be individually utilizing the entire cache and a larger cache reduces the number of conflict misses which will improve the overall hit rate of the cache.

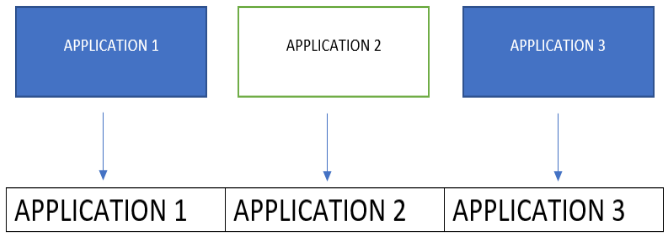


Figure 1: First Come First Serve Algorithm

ii) The Round Robin Scheduling Algorithm:

A Round Robin scheduling algorithm is one of the scheduling algorithms that improves the starvation problem caused by FCFS. This scheduling algorithm assigns time quantum (time slices of a particular portion). The applications in our project are run one after the other by caching one memory location from each application once. The response time of RR algorithm is better since this algorithm assigns an equal share of time to all the applications via assigning a fixed time quantum. This algorithm is starvation free, simple and easy to implement when all the applications are assigned in a circular order without assigning priority to the processes. This is also known as “cyclic execution”. The block representation of a typical Round Robin (RR) scheduling algorithm is as shown below:

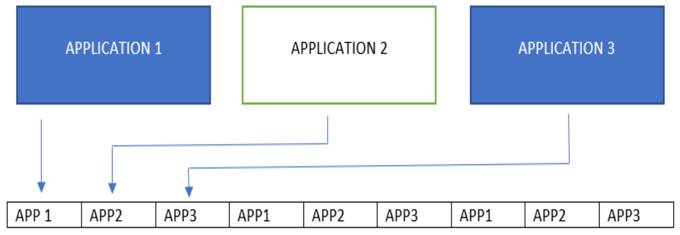


Figure 2: Round Robin Algorithm

III. SIMULATION SETUP

This section will describe the setup and process of the cache simulation for multiple applications on a single core using a unified cache. Firstly, the parameters are provided via command line. The trace files used are C Compiler trace, Spice trace and the DineroIV trace. The parameters provided are fetched by the C program and used to initialize the cache size, block size, cache mapping and cache organization. The cache size is kept fixed at 32KB while the block size is varied from 8 bytes to 32K bytes to analyze the performance of the cache. The cache mapping (direct mapping or full associative mapping) and cache organization (unified cache) are provided via command line parameters. The trace files required for the simulation are explicitly provided in the program. A file pointer is initialized for each trace file to be read. The number of blocks calculated (Number of Blocks = Cache Size / Block size), along with a pointer to the cache are used as arguments to setup and initialize the unified cache. The tag and valid bit for all blocks are initialized to zero during the cache setup. Also, a fifo counter is initialized to zero which is used during the fully associative cache mapping for FIFO replacement policy. For a Round-Robin scheduler algorithm each trace file is opened simultaneously in read mode and each memory address is read turn by turn (alike Preemptive scheduling) from each file till the EOF (End Of File) for all files is achieved, while for First Come First Served scheduler each trace file is read sequentially (alike Non-Preemptive scheduler) till the EOF (End Of File).

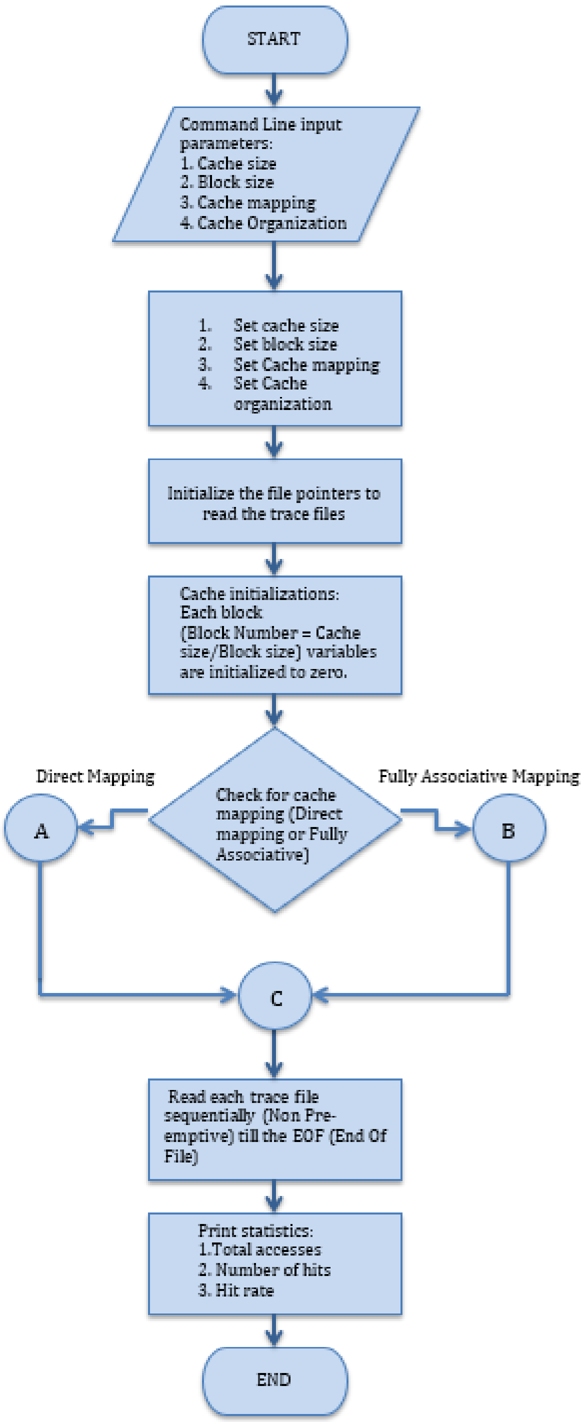

START
Command Line input parameters:
1. Cache size
2. Block size
3. Cache mapping
4. Cache Organization
1. Set cache size
2. Set block size
3. Set Cache mapping
4. Set Cache organization
Initialize the file pointers to read the trace files
Cache initializations: Each block (Block Number = Cache size/Block size) variables are initialized to zero.
Direct Mapping
Fully Associative Mapping
Check for cache mapping (Direct mapping or Fully Associative)
A
B
C
Read each trace file sequentially (Non Pre-emptive) till the EOF (End Of File)
Print statistics:
1.Total accesses
2. Number of hits
3. Hit rate
END

Figure 3: FCFS Scheduler Flow Chart

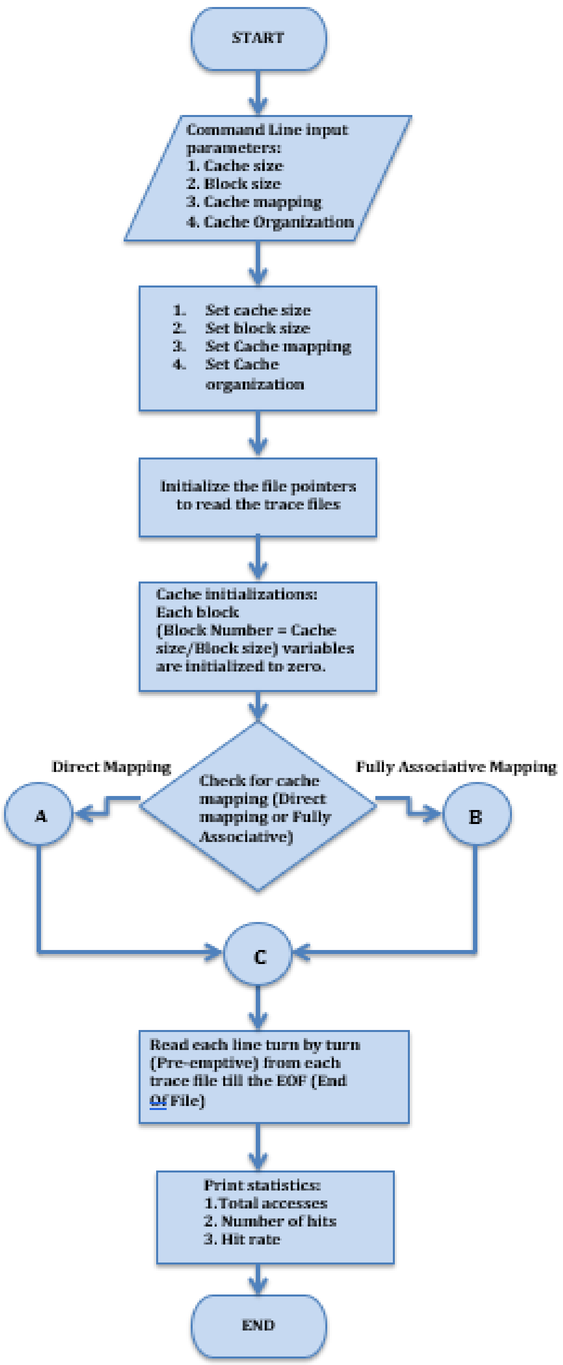

START
Command Line input parameters:
1. Cache size
2. Block size
3. Cache mapping
4. Cache Organization
1. Set cache size
2. Set block size
3. Set Cache mapping
4. Set Cache organization
Initialize the file pointers to read the trace files
Cache initializations: Each block (Block Number = Cache size/Block size) variables are initialized to zero.
Direct Mapping
Fully Associative Mapping
Check for cache mapping (Direct mapping or Fully Associative)
A
B
C
Read each line turn by turn (Pre-emptive) from each trace file till the EOF (End Of File)
Print statistics:
1.Total accesses
2. Number of hits
3. Hit rate
END

Figure 4: Round Robin Scheduler Flow Chart

In direct mapping, the counters for total memory address accesses (cache counter) and total number of hits (hit counter) are initialized to zero at the start of the function. The offset, index size and tag are extracted from the address fetched during the read. The extracted offset is matched with the block number in the cache. If it is matched, then the access counter is incremented. The process goes on till the offset is matched. After the offset is matched the program checks for a tag match. If it is not matched the tag gets updated at the block number where the offset was matched and also makes the valid bit = 1. If the tag is matched it will further check for a valid bit = 1. If valid bit = 0, the matched tag is updated with the new tag and valid bit is changed to 1 while if the valid bit is equal to 1 then the counter for hits is incremented by 1. Lastly, both the counters i.e. cache counter and hit counter are updated for printing the statistics which display the information regarding the total accesses done and total hits recorded for direct mapping

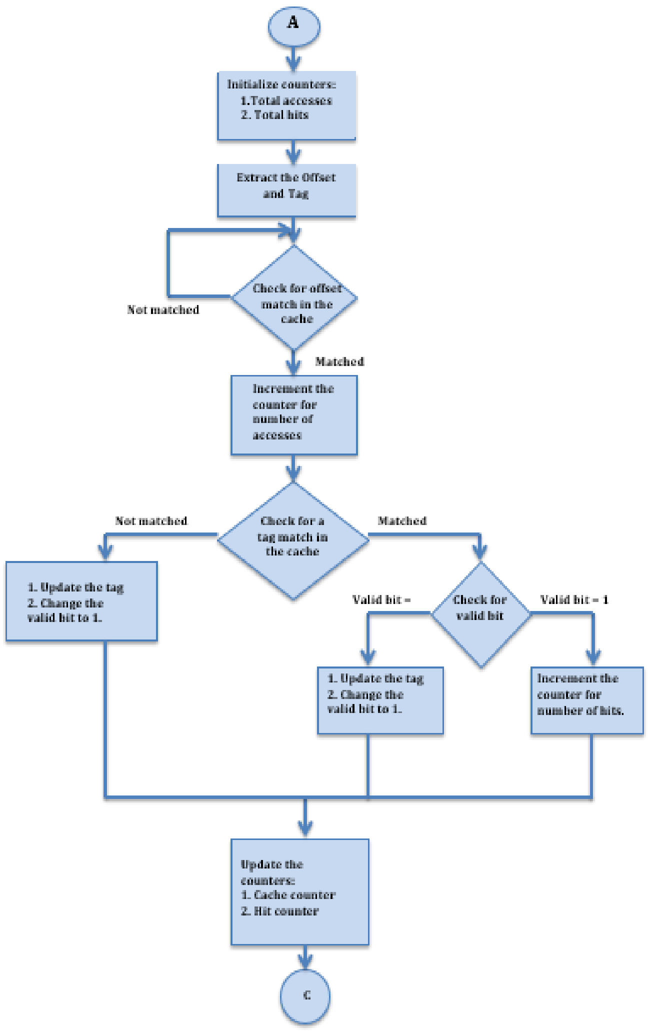

A
Initialize counters:
1.Total accesses
2. Total hits
Extract the Offset and Tag
Check for offset match in the cache
Not matched
Matched
Increment the counter for number of accesses
Check for a tag match in the cache
Not matched
Matched
1. Update the tag
2. Change the valid bit to 1.
Check for valid bit
Valid bit =
Valid bit = 1
1. Update the tag
2. Change the valid bit to 1.
Increment the counter for number of hits.
Update the counters:
1. Cache counter
2. Hit counter
C

Figure 5: Direct Mapped Cache Flow Chart

In Fully associative cache mapping, the counters for total memory address accesses (cache counter), total number of hits (hit counter), FIFO counter for incrementing the counter of all the other blocks which were not matched and a counter for replacing the oldest updated block (maximum counter), are initialized to zero at the start of the function. The offset and tag are extracted from the address fetched during the read. Then the cache is searched till the tag is matched and the valid bit is equal to 1. If a match is recorded the access counter (cache counter) along with hit counter and FIFO counter are incremented by 1 and the variable 'found' is updated to 1 which was initialized to zero at the beginning. The value of variable 'found' is used during the replacement policy. If the 'found' is not updated to 1, that means there was a miss and the cache needs to be updated for the tag to be stored. During the replacement, the oldest updated block i.e. the block which has the maximum counter is replaced with the new one. The tag is updated along with the FIFO counter increment and the valid bit is updated to 1. If the 'found' was updated to 1 the FIFO counter for the remaining blocks after the match was found is also incremented by 1. Also, during the cache search for matching the tag, if the block is found empty the tag is updated with the accessed tag and the valid bit is updated to 1 while the FIFO counter for that block is updated to zero which is used to specify that the block is most recently updated. Lastly, both the counters i.e. cache counter and hit counter are updated for printing the statistics which display the information regarding the total accesses done and total hits recorded for fully associative cache mapping.

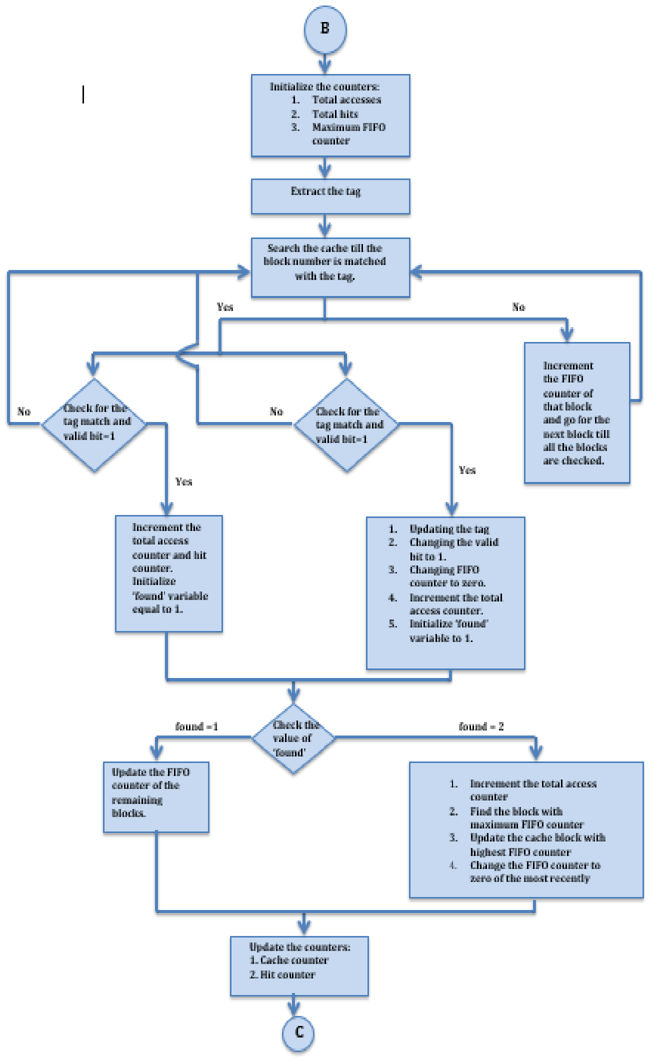
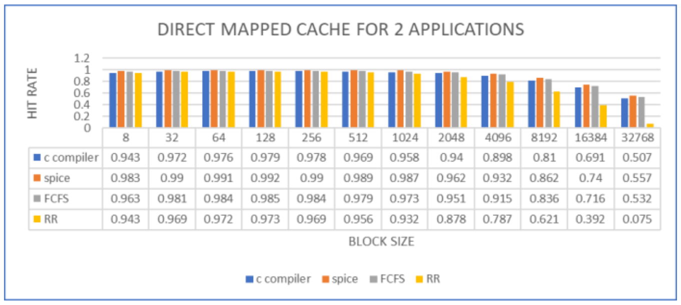

B
Initialize the counters:
1. Total accesses
2. Total hits
3. Maximum FIFO counter
Extract the tag
Search the cache till the block number is matched with the tag.
Yes
No
Increment the FIFO counter of that block and go for the next block till all the blocks are checked.
No
Check for the tag match and valid bit=1
No
Check for the tag match and valid bit=1
Yes
Yes
Increment the total access counter and hit counter. Initialize 'found' variable equal to 1.
1. Updating the tag
2. Changing the valid bit to 1.
3. Changing FIFO counter to zero.
4. Increment the total access counter.
5. Initialize 'found' variable to 1.
found =1
Check the value of 'found'
found = 2
Update the FIFO counter of the remaining blocks.
1. Increment the total access counter
2. Find the block with maximum FIFO counter
3. Update the cache block with highest FIFO counter
4. Change the FIFO counter to zero of the most recently
Update the counters:
1. Cache counter
2. Hit counter
C

Figure 6: Fully Associative Cache Flow Chart

IV. RESULT AND ANALYSIS

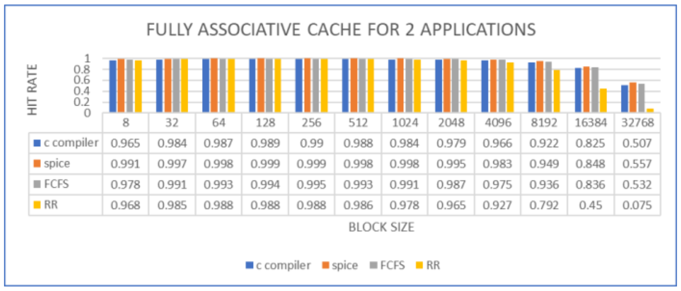
The main aim of the project is to determine the cache performance when multiple applications run at the same time. For this the two cache mapping techniques used are direct mapped cache and fully associative cache. The performance of each application for both the mapping techniques as well as for two and three applications which are cached at the same time is evaluated.in case of direct mapped cache we have one to one mapping indicating one memory location of the main memory maps to only one block in the cache. From the graph of the direct mapped cache, it is seen that the hit rate for both the c compiler trace and the spice trace is higher than that of the two applications cached together in Round Robin fashion. Also, as compared to first come first serve the hit rate for the two applications that are cached in the RR fashion is low. This is because the cache of the fixed size is now being used by two applications. Also, when the block size is varied from 8 to 32,768 bytes the hit rate for two applications is more for FCFS as compared to RR. In FCFS we have the one application run to completion and then then the other application utilizes the cache. Hence, at a given time only application is using the cache indication that the spatial locality of an application is fully exploited one at a time and hence the hit rate is higher. In case of RR, we have one instruction of one application that is scheduled followed by one instruction of the other application and so on. Hence the cache is shared making only half the cache available for one application and the other half for the other application. Thus, the number of conflict misses increases thereby reducing the overall hit rate of the system.



	8	32	64	128	256	512	1024	2048	4096	8192	16384	32768
c compiler	0.943	0.972	0.976	0.979	0.978	0.969	0.958	0.94	0.898	0.81	0.691	0.507
spice	0.983	0.99	0.991	0.992	0.99	0.989	0.987	0.962	0.932	0.862	0.74	0.557
FCFS	0.963	0.981	0.984	0.985	0.984	0.979	0.973	0.951	0.915	0.836	0.716	0.532
RR	0.943	0.969	0.972	0.973	0.969	0.956	0.932	0.878	0.787	0.621	0.392	0.075

Figure 7: Comparison of Cache performance for 2 apps

Similar results are obtained for fully associative cache as well. In case on of fully associative cache we have one to many mapping. Thus, one memory location can point to multiple blocks in the cache. Thus, when compared to direct mapped cache, the fully

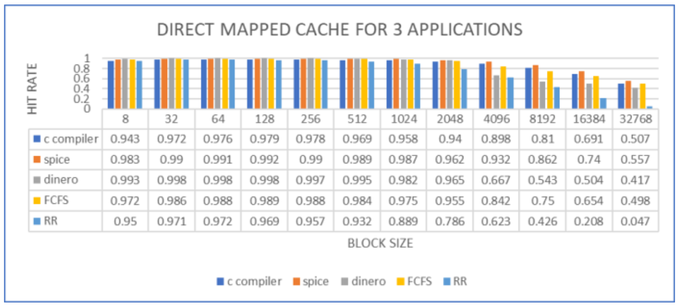


	8	32	64	128	256	512	1024	2048	4096	8192	16384	32768
c compiler	0.965	0.984	0.987	0.989	0.99	0.988	0.984	0.979	0.966	0.922	0.825	0.507
spice	0.991	0.997	0.998	0.999	0.999	0.998	0.998	0.995	0.983	0.949	0.848	0.557
FCFS	0.978	0.991	0.993	0.994	0.995	0.993	0.991	0.987	0.975	0.936	0.836	0.532
RR	0.968	0.985	0.988	0.988	0.988	0.986	0.978	0.965	0.927	0.792	0.45	0.075

Figure 8: Comparison of Cache performance for 2 apps

associative cache provides better performance.

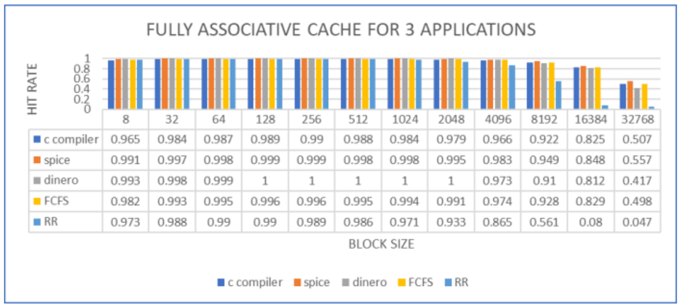
In case of three applications sharing the same cache, similar results are obtained to the above results. The third trace file that has been used is the Dinero trace file. From the graph of Dinero trace file it is evident that the hit rate for Dinero is very high. In some cases, the hit rate is almost 100%. Hence, when the other two applications are run along with Dinero, the overall hit rate increases as compared to two applications even though the cache is being shared by three applications. Thus, if a few applications out pf the multiple applications that run at the same time have a very high spatial locality the cache performance for all the applications improves significantly.



	8	32	64	128	256	512	1024	2048	4096	8192	16384	32768
c compiler	0.943	0.972	0.976	0.979	0.978	0.969	0.958	0.94	0.898	0.81	0.691	0.507
spice	0.983	0.99	0.991	0.992	0.99	0.989	0.987	0.962	0.932	0.862	0.74	0.557
dinero	0.993	0.998	0.998	0.998	0.997	0.995	0.982	0.965	0.667	0.543	0.504	0.417
FCFS	0.972	0.986	0.988	0.989	0.988	0.984	0.975	0.955	0.842	0.75	0.654	0.498
RR	0.95	0.971	0.972	0.969	0.957	0.932	0.889	0.786	0.623	0.426	0.208	0.047

Figure 9: Comparison of Cache performance for 3 apps

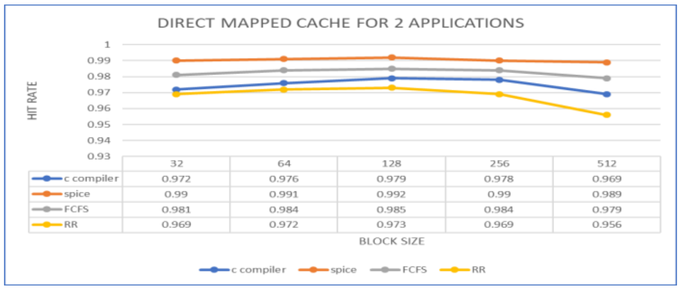
Just like two applications, fully associative cache further improves the overall cache performance as compared to direct mapped cache.



	8	32	64	128	256	512	1024	2048	4096	8192	16384	32768
c compiler	0.965	0.984	0.987	0.989	0.99	0.988	0.984	0.979	0.966	0.922	0.825	0.507
spice	0.991	0.997	0.998	0.999	0.999	0.998	0.998	0.995	0.983	0.949	0.848	0.557
dinero	0.993	0.998	0.999	1	1	1	1	1	0.973	0.91	0.812	0.417
FCFS	0.982	0.993	0.995	0.996	0.996	0.995	0.994	0.991	0.974	0.928	0.829	0.498
RR	0.973	0.988	0.99	0.99	0.989	0.986	0.971	0.933	0.865	0.561	0.08	0.047

Figure 10: Comparison of Cache performance for 3 apps

The performance evaluation is done for a block size of 8 bytes to 32,768 bytes which is the size of the cache. As seen, for direct mapped cache with increase in the cache size from 8 to 128 bytes the cache performance goes on improving. However, if the block size is increased beyond that, then the

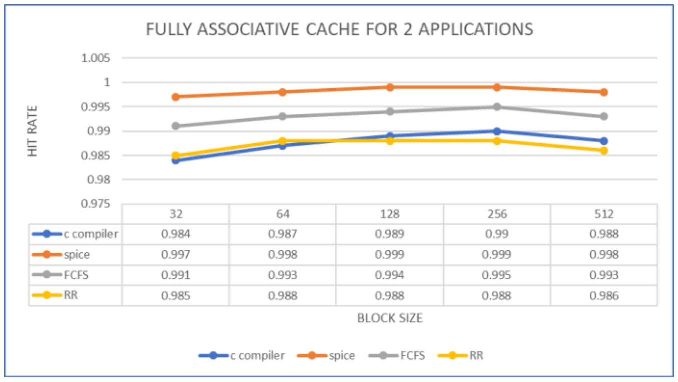


	32	64	128	256	512
c compiler	0.972	0.976	0.979	0.978	0.969
spice	0.99	0.991	0.992	0.99	0.989
FCFS	0.981	0.984	0.985	0.984	0.979
RR	0.969	0.972	0.973	0.969	0.956

Figure 11: Optimum Block Size for 2 apps

hit rate goes on reducing. The reason being that if the block size if large, the number of conflict misses goes on increasing. Also, another disadvantage of a very large block size is that the time taken to fetch the data from the main memory takes large. Hence the optimal block size for direct mapped cache for RR is 128 bytes.

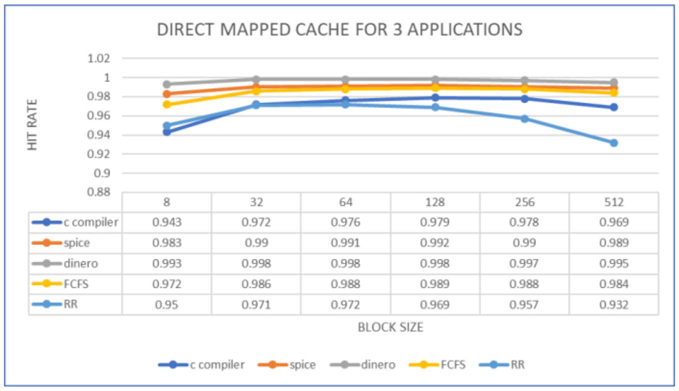
The same trend is observed for fully associative cache as well. 128 bytes is the optimal block size for two applications that are cached in a fully associative cache.



	32	64	128	256	512
c compiler	0.984	0.987	0.989	0.99	0.988
spice	0.997	0.998	0.999	0.999	0.998
FCFS	0.991	0.993	0.994	0.995	0.993
RR	0.985	0.988	0.988	0.988	0.986

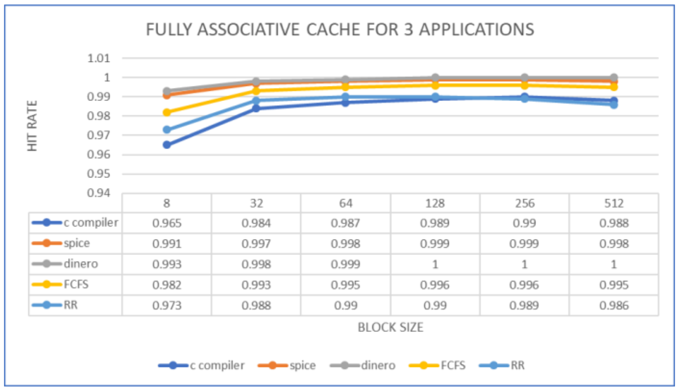
Figure 12: Optimum Block Size for 2 apps

For three applications as well, the same trend of 128 byte block size is observed for direct mapped as well as fully associative cache cahe.



	8	32	64	128	256	512
c compiler	0.943	0.972	0.976	0.979	0.978	0.969
spice	0.983	0.99	0.991	0.992	0.99	0.989
dinero	0.993	0.998	0.998	0.998	0.997	0.995
FCFS	0.972	0.986	0.988	0.989	0.988	0.984
RR	0.95	0.971	0.972	0.969	0.957	0.932

Figure 13: Optimum Block Size for 3 apps



	8	32	64	128	256	512
c compiler	0.965	0.984	0.987	0.989	0.99	0.988
spice	0.991	0.997	0.998	0.999	0.999	0.998
dinero	0.993	0.998	0.999	1	1	1
FCFS	0.982	0.993	0.995	0.996	0.996	0.995
RR	0.973	0.988	0.99	0.99	0.989	0.986

Figure 14: Optimum Block Size for 3 apps

V. LESSON LEARNED

The project analyzes the performance of the cache when multiple applications are run on a single core system. The results observed for two and three applications are scheduled using Round Robin (RR) and First Come First Served (FCFS) algorithms. The cache mapping functionalities used are direct mapped and fully associative cache. The cache replacement policy chosen is the First In First Out (FIFO) as this policy gives the worst effective performance as compared to the Least Recently Used (LRU) algorithm, hence analyzing this cache simulator for the worst possible case. The block size is varied with respect to hit rates in the cache. It is observed that Round Robin provides a good simulation time but the FCFS algorithm provides better spatial locality making it a good scheduling algorithm than the RR in this cache simulator design. The next observation is that the optimum block size for the cache is 128 bytes since after this block size, the hit rates go on decreasing since the conflict misses in the cache increase. Also when one application with the least hit rate is scheduled with multiple applications with higher hit rates, the total hit rate of the cache is increased.

REFERENCES

[1] Moinuddin K. Qureshi, Yale N. Patt, **“Utility based cache Partitioning: A Low-Overhead, High Performance, Runtime Mechanism to Partition Shared Caches”** , The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’06).